Johnson Grant
IN-61-CR
15145
p. 36

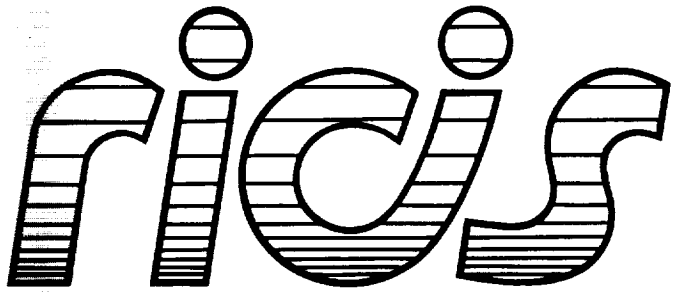
A RESEARCH REVIEW OF QUALITY ASSESSMENT FOR SOFTWARE

SofTech, Incorporated

April 30, 1991

Cooperative Agreement NCC 9-16
Research Activity No. SE.18
Deliverable 2.10A

NASA Johnson Space Center
Information Systems Directorate
Information Technology Division



Research Institute for Computing and Information Systems
University of Houston - Clear Lake

T·E·C·H·N·I·C·A·L R·E·P·O·R·T

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information systems in 1986 to encourage NASA Johnson Space Center and local industry to actively support research in the computing and information sciences. As part of this endeavor, UH-Clear Lake proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a three-year cooperative agreement with UH-Clear Lake beginning in May, 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The mission of RICIS is to conduct, coordinate and disseminate research on computing and information systems among researchers, sponsors and users from UH-Clear Lake, NASA/JSC, and other research organizations. Within UH-Clear Lake, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business, Education, Human Sciences and Humanities, and Natural and Applied Sciences.

Other research organizations are involved via the "gateway" concept. UH-Clear Lake establishes relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research.

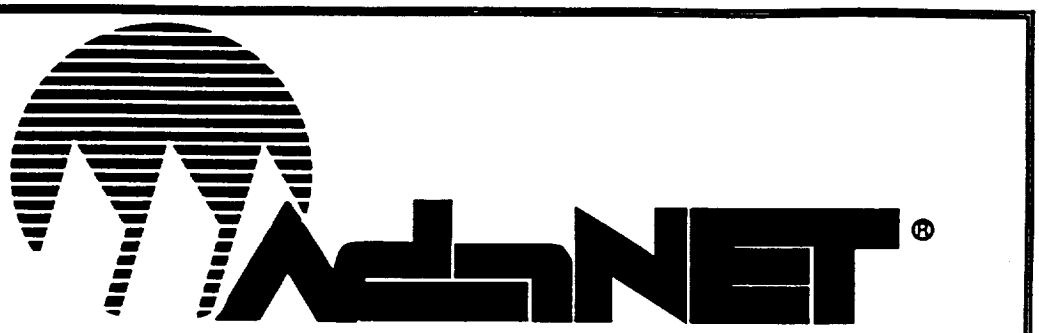
A major role of RICIS is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. Working jointly with NASA/JSC, RICIS advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research, and integrates technical results into the cooperative goals of UH-Clear Lake and NASA/JSC.

A RESEARCH REVIEW OF QUALITY ASSESSMENT FOR SOFTWARE

Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by SofTech, Incorporated. Dr. Charles McKay served as RICIS research representative.

Funding has been provided by Information Technology Division, Information Systems Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this activity was Ernest M. Fridge, of the Software Technology Branch, Information Technology Division, Information Systems Directorate, NASA/JSC.



A Research Review of Quality Assessment for Software

ADANET-FD-R&T-086-0

April 30, 1991

Subcontract No. 044
Cooperative Agreement NCC9-16
Project No. RICIS No. SE.18

Submitted to:
MountainNet, Inc.
P.O. Box 370
Dellslow, WV 26531-0370

Prepared by:
SofTech, Inc.
1300 Hercules Drive
Suite 105
Houston, TX 77058

TABLE OF CONTENTS

Executive Summary

This paper is the first of two papers which will recommend measures to assess the quality of software submitted to the AdaNet program. In this paper the quality factors that are important to software reuse are explored and methods of evaluating those factors are discussed. A follow-on trade study will recommend specific measures.

Quality factors important to software reuse are: correctness, reliability, verifiability, understandability, modifiability, and certifiability. Certifiability is included because the documentation of many factors about a software component such as its efficiency, portability, and development history, constitute a class of factors important to some users, not important at all to others, and impossible for AdaNet to distinguish between a priori.

The quality factors may be assessed in different ways. There are a few quantitative measures which have been shown to indicate software quality. However, it is believed that there exist many factors that indicate quality and have not been empirically validated due to their subjective nature. We characterized these subjective factors by the way in which they support the software engineering principles of abstraction, information hiding, modularity, localization, confirmability, uniformity, and completeness.

The development process has a major effect on the quality of software. The verification activities if they include correctness proofs or rigorous inspections, have been shown to reduce errors in software by more than an order of magnitude.

Methods for evaluating those factors affecting quality include both automatable procedures and those that must be manually applied. Only a few methods exist that are totally objective in the sense that they may be measured and once measured, compared to an empirically derived scale which gives their merit. Many methods may measure certain characteristics, but once having rendered a numerical value, offer no scale that qualifies that number. Some methods are strictly subjective.

1.0 Introduction

AdaNet is a reuse repository for software and its associated lifecycle products. Quality assessment for AdaNet purposes is to determine if the software and its associated lifecycle products are of sufficient quality such that a reuser would want to reuse them and, in fact, determine if the software and its associated products should be placed in the repository. This implies two points of qualification: first, upon the submittal of the product to the library; and second, by the reuser to determine if the product is of sufficient quality for it to be reused. Therefore, quality assessment must be addressed from the standpoint of:

- Assessing the quality of software and its associated products submitted to the library.
- Providing a reuser adequate assurance or measure of the quality of the software and its associated products.

This report, however, attempts only enlightenment as to what software quality is, the factors that influence software quality and the means of measuring those factors. A follow-on trade study will address the questions of which factors AdaNet should attempt to assess.

2.0 Software Quality

Software quality is the degree that software meets the quality specifications placed upon it. The same statement may be made for any product. The quality of any product may be perceived differently by different users, depending on the purpose they want the product for. Several factors of software quality are examined. Some of these factors are extensional, that is they are characteristics of the software itself; some are intensional factors that somehow characterize the process of producing software.

2.1 Characteristics of Software Quality

Having said that software quality depends on quality specifications, the next step would be to characterize how software may be specified. Unfortunately, there is a wide variation in opinion as to what these characterizations are. Table 2.1-1 shows some of the varied opinions on this.

[DEUTSCH]:	[2168], [McCall]	[BOEHM76]
Correctness	Correctness	Accessibility
Efficiency	Efficiency	Accountability
Expandability	Flexibility	Accuracy
Flexibility	Integrity	Augmentability
Integrity	Interoperability	Communicativeness
Interoperability	Maintainability	Completeness
Maintainability	Portability	Conciseness
Portability	Reliability	Consistency
Usability	Reusability	Device-Independent
Reliability	Testability	Efficiency
Reusability	Usability	Human Engineering
Safety		Legibility
Survivability		Maintainability
Verifiability		Modifiability
		Portability
		Reliability
		Robustness
		Self-Containedness
		Self-Descriptiveness
		Structuredness
		Testability
		Understandability
		Usability

Table 2.1-1 Various Quality Characteristics

That several of these quality characteristics can affect potential reuse of components has been recognized in previous and current efforts to reuse software. To distill which quality characteristics are important for reuse, the opinions of two industry giants who have long studied software and quality will be examined.

2.2 What Quality Characteristics Must AdaNet Judge?

In [CALDIERA] Victor Basili recognized the quality characteristics of correctness, readability (understandability), testability (verifiability), ease of modification and performance as important for component reuse.

Barry Boehm stated that the acquirer of a software package is mainly concerned with three questions [BOEHM76]:

- "How well (easily, reliably, efficiently) can I use it as-is?"
- "How easy is it to maintain (understand, modify, and retest)?"
- "Can I still use it if I change my environment?"

He then proceeded to stratify the qualify characteristics listed in Table 2.1-1 to address the above questions. His conclusions are shown in Figure 2.2-1.

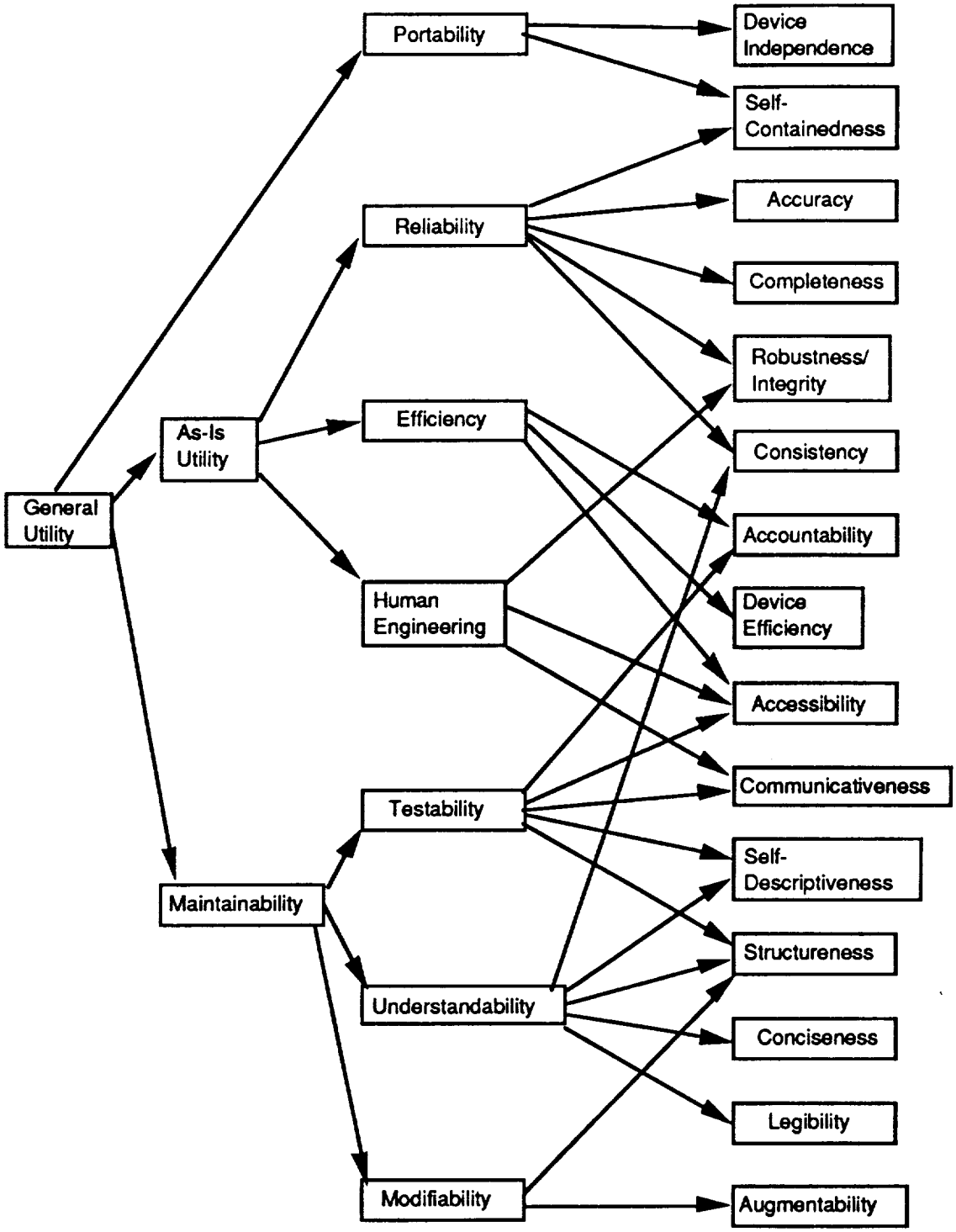


Figure 2.2-1 Software Quality Characteristics Tree [BOEHM76]

It can be seen that (allowing for the jargon) four of Basili's five reuse qualities (readability, testability, ease of modification, and performance) map directly to the the second tier of Boehm's quality characteristics tree. Combining the two lists (and unifying the common terms) gives:

Correctness
Reliability
Verifiability
Understandability
Modifiability
Human engineering
Efficiency
Portability

Combining the last three items on the list (for reasons to be discussed below) gives the quality characteristics that shall be considered (for this paper) important for software reuse.

Correctness
Reliability
Verifiability
Understandability
Modifiability
Certifiability (Human Engineering, Efficiency, and Portability)

Each of these terms is discussed below.

2.2.1 Correctness

A program may be considered correct if it will transform a specified input set of data into a specified output set of data. As measured, correctness is the degree to which software design and code will implement the software specification. Correctness implies a lack of errors in software. Correctness also implies that the software is complete in its specification and implementation of that specification.

Note that this definition does not imply that the specification itself is what the customer wanted. For reuse purposes, only the customer knows what they want and will make their own decisions about that. What must be given to them is assurance that the software correctly implements the provided specification.

2.2.2 Reliability

Reliability is the demonstrated ability of software to perform dependably. As measured reliability denotes failure or lack of failure of the software. Measurement of reliability may involve analysis of a distribution of errors detected during product development or the execution of the product.

To have good reliability software should exhibit a low failure rate. A low failure rate would seem to imply a low error rate. A low error rate is also a prerequisite for correctness. Similarly, both terms imply that the software performs according to its specification. However, reliability also connotes the idea of robustness,i.e., the tendency (or lack of) of the software to fail even when it is operating outside of the context that its specification is intended to address. This operational facet is what shall be emphasized when the term reliability is used.

2.2.3 Verifiability

Verifiability is the ease with which software may be tested and otherwise verified. Verifiability is concerned with several facets of the software and the associated products. Some of these aspects are extensional to the code such as how the software is structured. Others are intensional. How easily the software may be mapped to its requirements is an intensional factor since, without good requirements, the mapping cannot be evaluated (or will be poorly evaluated). Likewise, some of the aspects of verifiability are measureable, others are only subjectively assessable.

2.2.4 Understandability

Software and its associated products are understandable if their individual and combined purpose, function, and operation is clear to a reader. Understandability implies readability, consistency, and self-descriptiveness in the software and quality of documentation. Use of documentation and coding standards and simplicity in the design and code are major contributors to understandability.

2.2.5 Modifiability

Change can be considered normal for a component and thereby, so can the modification of the component. Quality assessment for reuse purposes must consider that most components will have to be modified in some fashion before being reused.

Any assessment of modifiability is strictly inferential in nature. However, other measureable factors influence modifiability and so may be used for assessment. For instance, if any change to a component is likely to introduce errors because of component complexity, that component should be viewed in a negative sense. Conversely, if a component is designed for easy modification it should receive a higher rating for reuse.

2.2.6 Certifiability

Certifiability consists of a number of quality characteristics rolled together primarily because AdaNet cannot assess them in an objective fashion. This lack of objectivity stems not so much from a lack of measurable factors and effects, but rather from an inability to assess the needs of the reuser. Efficiency and portability may be highly important for some applications but not the least important in others. A routine that is highly unportable may be ideal for a reuser who happens to be using the "right" environment. A routine that is not efficient may, nevertheless, have just the right interface for an engineer building a prototype where performance is not an issue.

However, this category forms a highly important means for a reuser to determine if a component is right for him. As indicated in the introduction, AdaNet must not only judge those quality factors that directly indicate a reuse potential for any component, but also those factors which may make a component attractive to a potential reuser. Prieto-Diaz and Freeman noted in [PRIETO_DIAZ] that "For code reuse to be attractive, the overall effort to reuse the code must be less than the effort to create new code." For AdaNet customers the cost of reuse must be balanced against the cost of development and testing of the new code. The development cost includes the cost of the designers, reviewers and programmers, and the cost of producing the documentation specific to that component

(these costs could be estimated and attached to each component in the library for a reuser to access when evaluating the component for his purposes).

Many projects that might consider the reuse of AdaNet components, have stringent (government/customer imposed) requirements for the quality of their products. AdaNet products that have a documented "quality pedigree" may offer powerful economic incentives for their reuse. That the correct engineering activities have been performed and documented, applicable standards applied to the software interfaces and project documentation, and all performance criteria and system dependencies have been verified and documented, can provide enormous incentive for a potential reuser.

Characteristics such as efficiency or portability should be documented. If system dependencies are limited or isolated then those modules that are system dependent, what the dependencies are, and which systems support the dependencies, should be noted for reuser prevue.

3.0 Factors Impacting Quality

Multiple factors have been shown to affect software quality. Please note that these factors are rarely quality characteristics themselves, but instead are attributes of the software have been shown or are believe to affect the software in a manner that enhances or detracts from the quality of a particular software component.

Deutsch and Willis in [DEUTSCH] addressed quality as consisting of product and process quality, product quality being an attribute of: documents, designs, code, tests; and process quality being an attribute of techniques, tools, people, organization, and facilities.

This section will consider a number of factors that influence software quality and will classify them into the product and process categories defined above. Presented will be quantitative data that empirically demonstrates the effects of some aspects of software products and processes upon quality. Also discussed will be non-quantitative criteria that is largely felt to affect software quality.

3.1 Product Specific Criteria

Product specific criteria include those assessments that can be made by code examination. In some cases the code examination can be automated since the specific criteria is objective and implementable. However, empirical data does not exist for all the information that may be revealed through code examination. This non-quantitative criterion is brought out in section 3.1.2 Engineering Principles.

3.1.1 Quantitative Factors

Below are presented factors which are measurable and whose measure has been shown to correlate to whatever degree with one or more of the software characteristics.

3.1.1.1 Complexity and Volume

Based on empirical data it appears that complexity and volume factors can affect the quality characteristics of correctness (and so reliability), modifiability, verifiability, and

understandability. Complexity and volume are commonly defined and measured by the McCabe and Halstead metrics which will be discussed later. It appears that these two measures can predict when software will:

- Exhibit more errors

  Walsh in [WALSH] reported on a military software project to which McCabe's metric was applied. They found 23% of the routines with a metric value greater than 10 accounted for 53% of all bugs. Walsh further stated that in the same study of 276 procedures, the routines with M greater than 10 and 21% more errors per line of code than those with metric values below 10. [BEIZER83] claims that Curtis substantiates these figures.

  In [AKIYAMA] it was shown that the number of errors in code were not as closely correlated with module length as it was with the sum of subroutine calls and decision points in the control structure (.92 correlation coefficient).

  In [FUNAMI] Funami and Halstead used Akiyama's data to show that the number of elementary discrimination required to generate a program correlated to the number of bugs with a correlation coefficient of .982. The number of elementary discriminations was computed using measures of program volume and potential volume [FUNAMI].

- Be more difficult to test

  Curtis, Sheppard, and Milliman in [CURTIS] claim, based on their experimental evidence, that "the software complexity metrics developed by Halstead and McCabe are related to the difficulty programmers experience in locating errors in code." They further state that "Code which is more psychologically complex may also be more error-prone and difficult to test."

- Inhibit modification

  Gibson and Senn reported in [GIBSON] that their studies "indicate that structural differences do impact performance", performance in this case being the ability to successfully modify the software. They concluded that complex programs are more difficult to maintain and require more corrective maintenance. Gibson and Senn also reported that Halstead's E, McCabe's v(G), Woodward's K, and Gaffney's Jumps appeared to relate to the time required to implement changes and Chen's Min and Benyon-Tinker's C2 appeared to relate to the rate of introducing errors during modification.

3.1.1.2 Size

Quantitative data on the effects of component size on the error rate is varied. [TAKAHASHI] reported that source code size had no demonstrated influence on the program error rate.

However, in [WITHROW] error statistics indicated that Ada packages of 200 to 250 lines had a minimum error rate when compared to other size packages. This inverse relationship has been noted in other studies with the optimal size ranging up to 500 lines of code. The range of mean error densities ranged from 1.8 to 8.3 errors per thousand lines of code.

At this time it appears no objective measure of the influence of code size can be made.

3.1.2 Engineering Principles

Examination of code can reveal a great deal more than the factors listed in the section above. Information about design, coding conventions and style, dependencies, default and initial values, error handling, can be seen in a code examination. This author was unable to find empirical evidence that show the effects of design method, coding style, etc., upon software quality.

However, this is not to say there is no evidence. Authoritarian evidence is rhetorically valid, particularly when there is a preponderance of opinions on the subject. Ultimately, all authoritarian arguments are based on experience and beliefs of those authorities.

An example of these authoritarian arguments regards the design method used in building software. Most authorities now seem to agree on the benefits for using object-oriented or object-based methods. Their argument may be based on reasoning like that used by Parnas [PARNAS] or on their own personal experience in using the methods. Regardless of the reasoning, most experts would now agree that object-oriented is a "good thing." Quantitatively assessing how much of a good thing it is, is another matter.

A list of the good things that software might exhibit would include minimal interfaces, client/server architecture, coding standards;however, some of the features that might have been considered good things in the past may not be as preferable, depending on the particular instance. Structured design is probably a design technique which, while still valuable, might be diminished in the eyes of some experts who would choose a later, more modern technique.

To not restrict ourselves to judgement criterion which may become quickly outmoded, an intermediate set of criteria to judge software will be used. Booch suggested that application of the principles of abstraction, information hiding, modularity, localization, uniformity, completeness, confirmability would be sufficient to engineer "good" software [BOOCH]. We will adopt these engineering principles as indicators for software quality and briefly discuss the applicability of each term based on Booch's discussion. To the extent that software exhibits these principles, we will consider that good. To the extent that software does not show or violates these principles, we shall consider that bad. It should be noted first that there does exist automated means of measuring the application of these principles in software. However, there exists no objective scale by which to assess a particular software component as good or bad based on the measure of one of these principles. In addition, while it is possible to render a number representing the application of these principles, this number may not represent the timely and purposeful application of these principles. Thus, subjective examination to assess the application of these principles may still be necessary.

3.1.2.1 Abstraction & Information Hiding

Abstraction is to emphasize the essential qualities while de-emphasizing the inessential details. Information hiding can be considered the hiding of those inessential details. Since the human mind can only consider a limited number of items simultaneously, these principles allow the mind to focus on what is important and pertinent and not become cluttered and confused with too much detail. Abstraction and information hiding are believed to aid correctness, modifiability, understandability, and efficiency.

Functions, processes, and data may be abstracted. A procedure is an example of functional abstraction, while the Ada task is an example of process abstraction. Objects are examples

of data abstraction. In each case the principles of abstraction and information hiding are used to permit a conceptualization and naming of a sequence of actions, a parallel action, or an entity in the problem space.

3.1.2.2 Modularity, Localization, & Confirmability

Modularity, localization, and confirmability deal with the structuring of the software. Modularity is the purposeful structuring of software towards some goal(s). The modularization of the program must necessarily reflect the top level abstractions that are made in the program design. We usually consider good modularity as exhibiting the properties of loose coupling and strong cohesion. Localization is the principle that facilitates loose coupling and strong cohesion. Modularity and localization are believed to aid modifiability, correctness, and understandability.

The principle of confirmability directly supports verifiability. Confirmability indicates that the software is structured so that it can be easily tested. This structuring has as much to do with the control structure of the software as with the decompositions that are used.

3.1.2.3 Uniformity & Completeness

Uniformity denotes a consistency in style of description. This consistency extends to both documentation and software. In documentation uniformity implies the use of standards, in code it implies a coding style. The standards and style used must be aimed at promoting understandability, the quality factor that uniformity supports.

Completeness insures that all required elements of documentation and code are present. For documentation this may mean complete listings, complete traceability, complete analysis, etc. For code completeness *it* implies that all specified elements are present and performed and that all preconditions such as initializations are included. Completeness supports the quality factors of correctness and understandability.

3.2 Development Process Criteria

Why examine the process used to produce software components? After all, we are seeking to determine the quality of the component not assess a software project. Surprisingly it turns out that the most significant factor(s) in producing quality software is the process used in developing the software.

This fact is due to the inevitable and unanimous (or as close to unanimous as can be) conclusion that no significant piece of software can ever be built without containing errors and can never be verified or tested so as to guarantee the complete absence of errors. Therefore, the production of software should minimize the number of errors in the software. By adherence to some of the methods discussed below, some front runners like Harlan Mills, have been able to produce software systems that have never failed and so must be considered nearly error free.

This side effect of rigorous development processes has not been unnoticed. It has been observed that there exists "an intimate relationship between the reliability of a product and the process used to develop that product." [IEEE982.2] In [HUMJ87] it is stated "The quality of a software product stems, in large part, from the quality of the process used to create it." As part of their efforts to improve software quality, the Software Engineering Institute (SEI) at Carnegie Mellon University has developed a categorization for contractors

[HUMJ87] [HUMS87] [HUM89] based upon assessing the contractor's development process (see section 4.2).

Many of the points raised in this report under process criteria and assessment are also brought out in the SEI report(s). Although the SEI report and this report agree on the importance the management and engineering processes for rendering quality, the focus of the SEI report and this one are different. The SEI report attempts to use qualification of the development processes as a way to gauge a contractor's ability to produce quality software for a government contract. This report attempts to use assessment of the development process to (indirectly) judge the quality of a software component.

3.2.1 Management

Management's role in determining software quality is to establish the framework necessary to support the engineering processes responsible for actually producing the products. The support includes process definition and resource management. These activities are discussed below. Without an adequate framework, the engineering process cannot hope to develop a successful or quality product.

The effect of management upon software quality is based on authoritarian evidence. However, the number of failed software projects that did not have well defined production processes attest to the necessity of good management activities and their effect. Recognizing that ultimately all aspects of software development, its success or failure, quality, etc. are economic activities, we see that the management of the software process is vital to good quality.

3.2.1.1 Development Models

The first development model for software has been termed the code and fix model. This development model produced poor quality software that was rarely adequate for the user's needs. Though several different forms of this model (such as the evolutionary model) were tried, its failures convinced the industry that development models for engineering software was necessary.

The development model that has shaped major system development (at least those of governmental agencies) for the past several years has been the waterfall lifecycle model. The waterfall lifecycle model offers a phased approach that provides intermediate points of assessment and review by the contracting agencies. This provides the contracting agencies decidedly better insight into the management and development processes and whether they are moving toward the desired goal in a productive fashion.

Within the waterfall development model structure several strategies have emerged. Top-down and bottom-up are two of the most repeated development strategies. Although bottom-up development may be superior in a few instances, such as developing device drivers, most sources would favor the top-down approach. Mills points out in [MILLS76] "The necessity of top-down development in large software systems is born out of bitter experience with top-down design and bottom-up development."

Improvements to the waterfall model have been suggested. Basili and Turner have proposed a top-down, stepwise refinement approach called iterative enhancement [BASILI75]. Iterative enhancement involves starting with a simple implementation initially

and then iteratively enhancing analysis, design and implementation as more knowledge is gained about the system.

Boehm and Papaccio report in [PAPACCIO] that Parento analysis applies to software. Parento analysis shows that 20 percent of the problems cause 80 percent of rework costs. They recommend that development and V&V focus on high risk areas. Boehm has proposed a new development model called the spiral model [BOEHM88]. The main purpose of the spiral model is to avoid document driven (waterfall) and code driven (pre-waterfall) development models which Boehm claims inevitably cause premature design decisions that later prove costly to undue.

While there is a great deal of evidence to support the inferiority of the code and fix development model, there is little comparative empirical data to suggest the superiority of any of the later models to each other. On any particular project, however, certain factors such as risk or schedule pressure may affect resource allocations within that project. This aspect is discussed below.

3.2.1.2 Process Models

It has been recognized that there are several activities that must be managed over the phases of the lifecycle such as configuration management, quality assurance, the development processes themselves, measurement of the development processes, and the associated documentation to name a few. Process models attempt to describe how to conduct these activities in a controlled and measurable manner.

Process models must define the methods, procedures, organization, measurement, documentation, tools, and training to achieve the desired ends. A small example will emphasize this point.

Fred Brooks contends that conceptual integrity is the most important consideration in system design [BROOKS]. Basili and Reiter in [BASILI79] compared the effectiveness of three types of teams: a single person team, a "disciplined" team consisting of three persons rigorously using a specified methodology, and an"ad hoc" team. The single person and ad hoc teams were allowed to develop software using methods of their own choosing. Their study found that disciplined teams produced smaller programs than the ad hoc team, but larger than that of the single person team. The authors also found that cyclomatic complexity of the programs examined indicated that dedicated teams using a specified approach were as effective and sometimes more effective than the single person team in producing less complex programs.They contributed this to conceptual integrity and noted,"Conceptual integrity certainly has an impact on the structural quality of the software being produced, resulting in a closer-knit design and implementation."

Thus in this instance, the use of a design methodology is shown necessary to achieve conceptual integrity as a quality goal for the design. As stated above, process models must define methods.

Another example is a development process that allows underdeveloped system documentation. Takahashi and Kamayachi reported in [TAKAHASHI] that design documents which lack sufficient detail can produce errors as well as documents that do not reflect system modifications. So a development process model that doesn't provide for rigorous review, updating, and correction of project documentation can result in errors in the developed system. Again, process models must define procedures.

While these few examples hardly exhaust the impact of software process models on quality, they do show that process models that are not rigorous can affect quality. Sufficient evidence to empirically demonstrate the superiority of some process models over others, has not yet been gathered. However, certain characteristics have been noted that are necessary for any process model to achieve its aims. The SEI report(s) [HUMJ87] [HUMS87] [HUM89] focus on the evaluation of these necessary characteristics for any software process model. These characteristics will be discussed later as the methods to assess a process model.

3.2.1.3 Resources

Resources are the people, time, and money (equipment) necessary to get the job done. In light of our previous discussion, resources may be defined as those factors necessary to the process model.

Producing software is a people activity. The quality of the people involved in the production of software can greatly affect that software. In [TAKAHASHI], the level of skill of the programmers was found to significantly influence the error rate.

The tools and training provided to the project personnel with which to conduct the engineering activities is also a factor believed to influence quality. The SEI reports cover these factors extensively in their attempts to qualify the development processes.

Organization and management of project resources can also affect the quality. In order to maintain the conceptual integrity (discussed above), Brooks cites Harlan Mills as having proposed a personnel organization that he calls the chief programmer team [MILLS71]. The chief programming team is compared to a surgical team; one man is the main player responsible for, in this case, designing, writing, and testing the program, while all other team members support him and remove all nonessential work from the chief programmer.

Time is also a resource. Of course, time means money. In fact, these two resources are highly related. Scheduling may be thought of as the timely expenditure of money (in terms of people to perform activities and the equipment and training that the people need) to meet the ultimate time constraint. Time, however, is not people. In his book, "The Mythical Man-Month" Fred Brooks details the effect of equating time and people. This effect is expressed allegorically in an industry parable as the "nine women can't have a baby in one month" phenomenon. Attempts to mismanage resources in this manner nearly always result in lowered productivity.

Most research in these areas attempts to correlate productivity with the management, organization, skill and training of project personnel and the tools provided to them. It is widely believed that these same factors play a significant role in determining product quality.

3.2.2 Engineering

This section examines the impact that the technical activities can have on product quality. It should be realized that the success of every activity covered in this section is primarily dependent on the application and rigor of the process model (see section 3.2.1.2 Process Models) used for the activity. Since process models were discussed under Management, they will only be mentioned in passing in the Engineering activities sections or when some unique aspect of the particular process model for that activity should be brought out.

3.2.2.1 Requirements, Specification, and Design

The lifecycle phases of requirements analysis and design can greatly affect the quality of the software and this effect is well documented. Below are a few instances of this.

- In [RUBEY] they also found that the greatest single cause (28%) of errors was the incomplete or erroneous specification of the program. In a breakdown of the specification deficiencies, Rubey, Dana, and Biché found that over half of the deficiencies were due to incomplete or incorrect design considerations and "indicate either deficiencies in or the absence of the verification of the program specification or program design, "

- In [PRESS] a number of industry studies (TRW, Nippon Electric, Mitre Corp., and others) indicate that between 50 and 65 percent of all errors or defects were introduced during the design phase of development.

- In [BASILI84] 48 percent of errors were attributed to incorrect or misinterpreted specifications or requirements. Design errors accounted for 22 percent of the errors.

This large influence on the correctness of the software emphasize the importance of these activities on software quality.

Factors contributing to the reduction of errors introduced in these phases have been the use of methods, supporting tools, training in the methods, and in other words, a process model supporting requirements gathering, requirements and design modeling and traceability. Testimonials to the effectiveness of commercial tools and method(and so process models that use these tools and methods) are many (although may be inflated and certainly could not be considered objective), but the comparative effectiveness of a particular tool or methods in reducing errors in these phases has not been proven [YADAV]. However, the inclusion of certain activities as part of the process model for these phases has proven to have dramatic effects on requirements and design quality. These certain activities are covered in section 3.2.2.3 V&V.

3.2.2.2 Coding

The activity of coding is the point where all software errors are actually codified. Russell cites data published by Caspers Jones in 1986 suggesting that the defect density of software ranged from 49.5 to 94.6 defects per thousand lines of code [RUSSELL]. Many other studies substantiate these numbers (within a statistically believable variance). However, as we have seen in the previous sections (and also the ones to come) errors are actually introduced in many ways. Among the other things influencing software quality are the tools that are used to code.

3.2.2.2.1 Language and other tools

Since AdaNet is intended (at the time of this writing) to contain Ada software and its associated lifecycle products, the advantages that are attributed to Ada can extend to the AdaNet components. These advantages are many. The strong typing of Ada has been shown to prevent many classes of errors while the package structure has been proven to promote modifiability and reusability. A complete list of benefits provided by Ada are too many to list here and is not relevent anyway since, as stated above, no other languages need be considered.

Other tools used in the coding process, such as debuggers, have been shown to affect productivity [DUNSMORE], but no study found by this author has quantitatively assessed their effect on quality. The importance of modern programming tools on software quality, however, has been underscored by the inclusion in the SEI report [HUMPS87] of assessment concerning the use of tools on software projects.

3.2.2.2.2 Reuse of Software Parts

Another type of tool for software production (and the one AdaNet is intended to serve) is the reuse of software. Can the use of this type of tool affect the quality of the final product? Takahashi and Kamayachi reported in [TAKAHASHI] that the percentage of reused (modified) modules in a program had no effect on the error rate. In [BASILI84], in a system where 72 percent of the code segments qualified as reused modules, 49 percent of all system modules containing errors were reused modules. This indicates that reuse of software can actually reduce errors.

3.2.2.3 V&V

In [RUBEY] the authors state, "Program development is a very error-prone activity, but approximately 98 percent of all errors are uncovered through normal program development efforts and only 2 (sic) percent of the errors remain to be found during the validation phase." They also state,"there is no single reason for unreliable software, and no single validation tool or technique is likely to detect all types of errors." However, their observations caused them to note that, "the verification of the program specification and design in advance of coding and debugging is a very beneficial activity and, indeed,is probably essential if reliable software is desired."

The above statement underscores the empirical results of the effects that these activities can have on software quality. There are three types of V&V activities that we will examine: review (inspections), proving correctness, and dynamic testing.

3.2.2.3.1 Formal and Informal Reviews

Formal reviews as specified in present day lifecycle model [MIL2167] [MIL2167A] [MIL2168] [IEEE1028] have been recognized as highly effective in promoting software correctness, understandability, and modifiability. However, rigorous informal reviews have produced some almost unbelievable results. A few of these are listed below.

- [GILB] reported that Project Orbit, a 500,000 line networked operating system developed by IBM, was delivered early and had about one hundred times fewer errors than would normally be expected using the inspection process.

- In [PRESS] a number of industry studies (as mentioned above) indicated that between 50 and 65 percent of all errors or defects were introduced during the design phase of development. However, Pressman also cites [JON86] as having shown that formal review techniques have been up to 75 percent effective in uncovering design flaws.

- Russell provides empirical evidence obtained on more than 300,000 lines of code (and its design) that indicate that inspections are two to four times more effective than formal designer testing or system testing [RUSSELL]. Russell indicates he has seen data that, depending on the test environment's complexity, code inspection can be up to 20 times more effective than testing.

The above results have been repeated in many instances. In [ACKER] they concluded, "while inspections do not eliminate testing, they can significantly reduce the testing effort because inspections are from two to 10 (sic) times more efficient at defect removal than testing."

Russell attributed the success of inspections to the types of errors that inspections are apt to uncover. Russell classified the defects reported for 2.5 million of inspected code according to the categories of whether they were on account of wrong, extra, and missing statements. He found that extra and missing statements accounted for over 50 percent of the errors found by the inspection process and noted, "Extra statements are more likely to be found only in inspections; it's almost impossible to find superfluous code through testing. Even missing statements are difficult to detect in testing, since omissions are often related to infrequent failure paths that are difficult to create in the product under test."

The inspection process described is a rigorously defined process first defined by Fagan [FAGAN]. Since that time many different sources have shown that modifications of the process can be made so long as the rigor and preparatory steps to the inspection are maintained [BISANT] (and many others).

3.2.2.3.2 Formal Verification

Formal program proving is advocated by many of the industry giants such as Dijkstra. When used, these methods have produced results as effective as those of the inspection method described above. Cobb and Mills in [COBB] state data that indicates that "functional verification leaves only two to five fixes per thousand lines of code to be made in later phases, compared to 10 to 30 fixes left in unit testing by debugging."

This result is perhaps true because despite its "mathematical" nature, formal proving is another form albeit intense of inspection. Despite the promise of its definitiveness, formal program proving will not ensure the errorless program. In [RUBEY] the number of specification errors led them to conclude that, "the ability to demonstrate a program's correspondence to its specification does not justify complete confidence in the program's correctness since a significant number of errors are due to an incomplete or erroneous specification ."

Despite this disclaimer, formal verification must be considered a premier technique for promoting software correctness.

3.2.2.3.3 Testing

Software testing is the oldest method of attempting to provide software correctness and reliability. Despite all the newer methods for assessing software, proper testing still remains essential. Russell notes that dynamic testing is better (than inspections) at finding problems relating to execution, timing, traffic, transaction rates, and system interactions [RUSSELL].

The number of specific methods of testing have increased greatly in the past twenty years. However, no methods or group of methods has emerged superior for all or a majority of situations. Most methods generally fall into one of two categories: black box or white box methods. Black box (functional testing) and white box (primarily path and branch testing) methods remain the most prevalent means for performing validation and verification testing and both types of testing are usually considered necessary. A recurring goal is to to

automate the testing activity, but no generally applicable method of doing this has been achieved.

Test planning has also evolved and presents many different approaches. The most commonly used strategies for test coverage have evolved out of the branch and path testing techniques. Planning for this type of test strategy was an economic activity as expressed in [IEEE982.2], "dynamic tests, the main tool for validation, have to be used cost-effectively with measures to provide the widest test coverage of a product." McCabe's metric has been applied as an attempt to measure the extent of this coverage (discussed later). Other more recent methods of testing avoid this path and branch planning and rely on the detection of failures rather than errors as the main reason and criteria for testing. A usage testing strategy allocates the testing resources in accordance with the probability that a failure is observed by the average user. In [COBB] Cobb and Mills demonstrated that 1.4% of the errors fixed in their study accounted for 53.7% of the failures. Fixing these 1.4% of their errors resulted in doubling the reliability of their product. Based on their results they concluded, "Statistical usage testing is 20 times more cost-effective in finding execution failures than coverage testing."

4.0 Methods for Quality Assessment

[IEEE982.1] defines:

- Six categories of product measures:

  1) Errors; Faults; Failures - Count of the number of defects with respect to human cause, source code bugs, observed system malfunctions.

  2) Mean-Time-to-Failure; Failure Rate - Derivative measures of defect occurrence and time.

  3) Reliability Growth and Projection - The assessment of change in failure-freeness of the product under test and in operation.

  4) Remaining Product Faults - The assessment of fault-freeness of the product in development, test, or maintenance.

  5) Completeness and Consistency - The assessment of the presence and agreement of all necessary software system parts.

  6) Complexity - The assessment of complicating factors in a system.

- Three categories of process measurements to the activities of development, test, and maintenance:

  1) Management Control - The assessment of guidance of the development and maintenance processes.

  2) Coverage - The assessment of the presence of all necessary activities to develop or maintain the software product.

  3) Risk, Benefit, Cost Evaluation - The assessment of the process tradeoffs of cost, schedule, and performance.

Specific counted and computed measures are given in [IEEE982.1] that address each of the above categories. However, as indicated in the preceding sections, this study shall not be limited to only quantitative measurement. [IEEE982.2] states:

> "Measurement is the comparison of a property of an object to a similar property of a standard reference. Measurements are effective when they are used either to orient decisions, to define corrective actions, or to get a better understanding of casual relationships between intended expectations and observed facts."

Since the use of simply counted and nonquantitative (subjective) data can similarly shape decisions about the assessment of quality, this type of data will also be used as a measurement.

This study will again use the broad categories of product and process measures to classify the various measures. Measures listed inside each of the product and process category constitute a superset of the ones listed above.

Measurement methods to be discussed in the following sections were gathered from several sources. To allow more immediate referral to the source of that measure, some special conventions will be followed. For measurements defined by [IEEE982.1] the measure number as it appears in section four of [IEEE982.1] will be given. For instance [4.40] would refer to measure 40 of section four. Other references will be given as appropriate and will use our standard bibliographic notation. Note that this referral does not necessarily denote the original source of that measure, but rather a location where the measure may be referenced.

Again, it is mentioned that this is an overview of how certain categories may be measured and not a endorsement of any particular method. A follow-on trade study will recommend specific measures for use in AdaNet quality assessment.

4.1 Product Assessment

Product assessment is based on examination of the software. Both objective and subjective measures are considered. Product assessment is divided into static, dynamic, and operational assessment. Operational assessment is the capture of failure data reported by the users. Dynamic assessment is performed by repository personnel.

4.1.1 Static Assessment

The most basic static assessment for software is to compile the code. The Ada compiler will catch many types of errors and form the "entry" level test for submitted software. Beyond this the code should be assessed in terms of the factors and principles discussed above.

Correctness proofs, based on the evidence presented above, would be a premium form of static assessment for any submitted component. However, since the virtues of correctness proving have been extolled above and would be the same if AdaNet were to perform the correctness proof, they will not be repeated in this section. Instead, correctness proving, if performed by the developer, will be considered as a means of assessing the engineering processes used by the submitter organization to enhance the component quality.

Inspections of the code (and documentation) by the AdaNet organization is discussed below as a means of static assessment.

4.1.1.1 Complexity & Volume

Basili suggests the metrics of volume, regularity, complexity, and reuse frequency for identifying candidate reuse components. He recommends using Halstead's Science Measures to measure volume and regularity [4.14] and McCabe's complexity measure [4.16] to judge code complexity [BASILI80-2][CALDERIA].

Based on the evidence presented earlier, Halstead's Science and McCabe's complexity measures would seem to be the definitive metrics for objectively predicting the quality factors of correctness, modifiability, and verifiability.

4.1.1.2 Abstraction & Information Hiding

User defined types, private and limited private types (with packages), tasking and its rendezvous, and exceptions are the main features offered by Ada to promote abstraction and information hiding. See [BOOCH] for a more complete discussion on these features.

A simple count of the declaration of private and limited private types together with packages to define objects, provides one measure of the use of abstraction and information hiding. These counts could also be compared to the number of non-private types declared.

The number of declarations of user defined types, user defined exceptions, the use of named constants, and the overloading of operators is another measure of abstraction, although probably not as strong as counting objects.

A ratio of the exported types, objects, and operations in a specification to the types and objects appearing only in the body would be a measure of information hiding.

Although the above counted measures could offer some quantitative feel for abstraction and information hiding in a component, the only way to judge if these principles were effectively implemented is by inspection.

4.1.1.3 Modularity,Localization, & Confirmability

McCabe advised partitioning routines whose metric exceeded ten [McCABE]. This suggestion has quantitative merit based on the findings presented above that demonstrate an increased error rate in routines with a complexity measure larger than ten.

Another measure of modularity is Design Structure [4.19] which can be used to assess a detailed design. To assess localization, those modules containing system dependencies could be checked to insure that they are isolated from other parts of the program.

Basili has proposed using data binding as a statistic to measure program quality [BASILI75]. Data binding occurs when a procedure or function modifies or accesses a global variable. Data flow complexity [4.25] is an example of one measure that assesses data binding. One of the main goals of the data binding measures is to reflect the use of global variables. Global variables can be tremendous propagators of runtime errors. Some measures simply count the number of global variables. Object oriented designs may influence future measures of data binding since only an objects methods (to borrow a term from from object oriented programming) may reference and modify an object. So in a

sense there is no global referencing, only the 'withing' dependence on an Ada package and its exported methods. Gannon, Katz, and Basili in [GANNON] discussed various metrics that might be used to assess Ada packages and how they could be used to characterize the structure of an Ada program.

As noted repeatedly by Harlan Mills and others, the intent behind Dijkstra's original concept of structured programming was to enhance the testability of the code. Examining the structure of the code remains a good way to assess confirmability. This would include checking things like:

Modules having only one exit.
Loops having only one exit.
No go to's.
Appropriate use of the types of loops and branching

Again, although the above tools can render numerical data about a component, the only way to determine if these principles were effectively used is by inspection.

4.1.1.4 Uniformity & Completeness

In some part, the application of these principles is aimed at the understandability or readability of the software. In [CALDIERA] it states, "The costs to reuse the component can be influenced by the readability of a code fragment, a characteristic that can again be partially evaluated using volume and complexity measures as well as measures of the nonredundancy and structuredness of the component's implementation." Thus we see yet another use of McCabe's complexity and the Halstead Science measures.

At this point let us note that these principles could also be applied to documentation. Some automated grammar checkers are offering measures of readability and interest. Other types of methods, such as the ones used for measuring the reading difficulty level in children's school books , could also be used on documentation.

Coding style deals with indentation and blocking; inline commenting for self descriptiveness, naming conventions for objects, loops, blocks, types, subprograms, tasks, etc.,which are ways of improving readability. However, some of the conventions of a coding style are necessary for proving correctness. Such things as default initializations and mode parameters, boolean expression, use of structured types, must be defined as preconditions in order to allow a proof.

Since automated checkers would probably only be appropriate with a particular coding style, three methods of assessment present themselves. One, obtaining the code checker from the component supplier and use it. Two, using a tool that would confirm that some attempt at code styling had been done, regardless of the particular conventions. Three, manually verifying a coding convention(s) had been used and confirming that those conventions met some minimum standard.

4.1.2 Dynamic Assessment

Dynamic assessment of software can be made only if the tests and expected test results for that software are available (unless tests are to be generated). In such a case the tests could be run and the results assessed against the results provided by the submitting organization.

Execution analyzers can be run with the test procedures to capture the actual execution paths traversed by the tests. The results may be used to verify test coverage measures.

If the component to be added to the repository can be captured as part of a larger program, then measures such as the ones described by Denson and Hooi in [DENSON] can be used. These methods assess the component versus environmental configurations that it operates in as a result of being embedded in a larger program that creates the environment. Here the term environment is used to describe the objects and their values existing at component execution. A comparison is made between the environments that the component is proven to operate in successfully to a first order approximation of the total possible environments. This measure may be used with functional testing as well.

Verifying performance measures is possible only if the necessary environment to create the specified load, drive the tests, and measure the result are available.

4.1.3 Operational Assessment

All operational assessment is based on the failures of the software as experienced by the (re)users. Operational assessment is the chief way to determine reliability. Some strategies, such as seeding, to test the effectiveness of the engineering processes to locate errors and the associated measures like Estimated number of faults remaining [4.22], Testing sufficiency [4.29], and Test accuracy [4.36] try to predict reliability. Musa claims that failure intensity can be determined during testing [MUSA90]. Failure intensity is usually measured in failures per execution time. In the testing process, measurement of test failure is usually expressed with metrics such as Run reliability [4.18]. Usually, it is only as software fails in the field that accurate reliability estimates may be made.

The definitive and foremost source for software reliability is Musa [MUSA75] [MUSA80]. [IEEE982.1] lists many measures that may be used to represent the reliability of the software. These include:

Fault density [4.1]
Cumulative failure profile [4.3]
Mean time to discover the next K faults [4.20]
Software purity level [4.21]
Residual fault count [27]
Failure analysis using elapsed time [4.28]
Mean-time-to-failure [4.30]
Failure rate [4.31]
Combined HW/SW system operational availability [4.39]

4.2 Process Assessment

[IEEE982.1] lists several computed metrics for assessing the management and engineering processes. These metrics will be listed as appropriate.

Also included will be a description of the methods used by the SEI for assessing and classifying the quality of the process models used by contractors to develop software. The SEI report [HUMS87] defines five levels of process maturity to characterize contractors:

1) Initial - "The initial environment has ill-defined procedures and controls. The organization does not consistently apply software engineering management to the process, nor does it use modern tools and technology. Level 1 organizations may have serious cost and schedule problems."

2) Repeatable - "At Level 2, the organization has generally learned to manage costs and schedules and the process is now repeatable. The organization uses standard methods and practices for managing software development activities such as cost estimating, scheduling, requirements changes, code changes, and status reviews."

3) Defined - "In Level 3, the process is well characterized and reasonably well understood. The organization defines its process in terms of software engineering standards and methods, and it has made a series of organizational and methodological improvements. These specifically include design and code reviews, training programs for programmers and review leaders, and increased organizational focus on software engineering. A major improvement in this phase is the establishment and staffing of a software engineering process group that focuses on the software engineering process and the adequacy with which it is implemented."

4) Managed - "In Level 4, the process is not only understood but it is quantified, measured, and reasonably well controlled. The organization typically bases its operating decisions on quantitative process data and conducts extensive analyses of the data gathered during software engineering reviews and tests. Tools are used increasingly to control and manage the design process as well as to support data gathering and analysis. The organization is learning to project expected errors with reasonable accuracy."

5) Optimized - "At Level 5, organizations have not only achieved a high degree of control over their process, they have a major focus on improving and optimizing its operation. This includes more sophisticated analyses on the error and cost data gathered during the process as well as the introduction of comprehensive error cause analysis and prevention studies. The data on the process are used iteratively to improve the process and achieve optimum performance."

The SEI report also classifies software development tools into two categories: Inefficient and Basic.

The SEI report lists well over a hundred questions with Yes/No answers to provide for the assessment and classification of the software development process used by an organization into the above categories. This study will not reiterate all of the SEI report, but will instead indicate a category of questions addressing some concern, where appropriate, and reference the SEI report, [HUMS87].

4.2.1 Management Process Assessment

Management activities include defining the processes that engineer's must execute to develop the products, ensuring that those processes are supported (resources), and that all (sub)processes integrate into a larger process (development model).

To evaluate the management process, management documentation (some project specific and some related to the developer organization) must be assessed. This would include software development plans, configuration management plans, quality assurance plans,

acquisition plans, standards and procedure document, etc. Each document must be evaluated by the processes it defines. Each process definition must describe:

production activities
verification activities
control activities
process measurement and evaluation activities

The sum of these activities should span all phases of development. The description for each activity must include:

inputs and outputs
supporting methods and procedures
supporting standards and tools
appropriate experience and training

The software development plan should be examined to verify the smooth integration of all processes and the organizational structuring detailing the allocation of responsibility to the project personnel.

The SEI report, [HUMS87] provided more specific questions addressing the above-listed topics.

Review of end-of-project documentation may reveal characteristics about management during the project. An example is schedule slips and management's reaction to such slips [BROOKS]. Does management panic and try to reallocate resources or will they remain dedicated to the software process and accept some delays?

There are some tools and measure which may be used to verify management decision. COCOMO [BOEHM81] can be used to verify cost estimates while the Required Software Reliability [4.33] measure can be use to assess the allocation of project resources and activities to high risk areas.

4.2.2 Engineering Processes Assessment

Assessment of the engineering processes would, in part, consist of determining that the activities defined by the management processes were successfully carried out. This would include examination of project documentation. For instance, requirements documents should be assessed for completeness and traceability since these can indicate how carefully the requirements were prepared and verified. Use of several measures described by [IEEE982.1] would be indicated here. Such measures include:

Requirements completeness [4.35]
Requirements compliance [4.23]
Number of conflicting requirements [4.12]
Requirements traceability [4.7]
Cause and effect graphing [4.6]

Evaluation of software documentation should assess their adequacy for use in a software maintenance environment [4.32]. Documentation should also be assessed to verify the component's behavior is described by the documentation [RAPIDSTD]

This assessment should also include project records, if accessible, such as software development folders and review minutes. A justification is provided by Mills in [MILLS76] where he states that, "A well-designed system of deep simplicities has a development history which is sharply distinguished from a brute-forced bowl of spaghetti. The most noticeable difference is the debugging history. A well-designed system can be put together with few errors during its implementation. A bowl of spaghetti will have a history of much error discovery and fixup." Examination of the review minutes on the design and the resulting modifications could reveal just such facts.

However, much of the evidence presented in section 3.2.2 indicates that some engineering activities, namely V&V activities, should be assessed in greater detail because of the profound effects they have on product quality. In particular, any assessment must ask:

Were verification activities carried out on the requirements and design activities?

Did the verification activities include correctness proofs and/or inspections?

Were the inspection steps rigorously defined and performed?

Besides these yes/no questions (that reflect, in part, the SEI evaluative criteria), there are measures that may be indicative of the effectiveness of the verification process . Mills suggests in [MILS76] that error-days be used as a measure of software quality. He says that the error-day measure "indicates probable future error incidents, but also indirectly indicates the effectiveness of the design and testing process. This measure corresponds to Fault-days number [4.4]. Used as a process measurement, fault-days number could judge the effectiveness of the verification processes for error detection.

Assessment of testing should examine test planning and address both validation and verification testing. All test results by the submitter organization should be evaluated to establish if the testing process was rigorous and sufficient resources were devoted to the activity.

Assessment of test coverage for validation testing could use indicators such as the [IEEE982.1] measures, Functional test coverage [4.5], and Test coverage [4.24]. An entity-based approach to assessing validation test coverage is discussed in [DENSON] (and above in section 4.1.2 Dynamic Assessment).

Verification test coverage may be assessed using McCabe's complexity metric. McCabe states in [McCABE] that if the complexity v [4.15, 4.16] of a program is greater than the number of paths tested, then one of the following must be true:

1) There is more testing to be done (more paths to be tested).

2) The program flow graph can be reduced in complexity by the difference.

3) Portions of the program can be reduced to inline code.

See [BEIZER83] for a definitive explanation of these techniques.

5.0 Sources

5.1 Published Standards on Quality Issues

[MIL2168] MIL-STD-2168
Software Quality Evaluation

[IEEE730] ANSI/IEEE Std. 730-1984
Software Quality Assurance Plans*

[IEEE982.1] IEEE Std 982.1-1988
Standard Dictionary of Measures to Produce Reliable Software*

[IEEE982.2] IEEE Std 982.2-1988
Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software (IEEE Std 982.1-1988)*

[IEEE983] ANSI/IEEE Std. 983-1986
Software Quality Assurance Planning*

[SMAP] Information System Life-Cycle and Documentation Standards, NASA Office of Safety, Reliability, Maintainability, and Quality Assurance, Software Management and Assurance Program (SMAP)

5.2 Related Standards

[IEEE1008] ANSI/IEEE Std. 1008-1987
Software Unit Testing*

[IEEE1012] ANSI/IEEE Std. 1012-1986
Software Verification and Validation Plans*

[IEEE1028] ANSI/IEEE Std 1028-1988
Standard for Software Reviews and Audits*

[IEEE1063] ANSI/IEEE Std. 1063-1987
Standard for Software User Documentation*

IEEE829] ANSI/IEEE Std. 829-1983
Software Test Documentation*

[MIL2167] MIL-STD-2167

* contained in *Software Engineering Standards*, Third Edition, IEEE, Oct. 1989

[MIL2167A] MIL-STD-2167A
Defense System Software Development

[MIL1521B] MIL-STD-1521B
Technical Reviews and Audits for Systems, Equipments, and Computer Programs

[MIL499A] MIL-STD-499A (USAF) Engineering Management

[SEM] System Engineering Management Guide
Defense Systems Management College, Fort Belvoir, Virginia

5.3 Other References

[ACKER] A. Ackerman, L. Buchwald, and F. Lewski, "Software Inspections: An Effective Verification Process", IEEE Software, May 1989

[ADAMAT] Ada Measurement and Analysis Tool (AdaMAT) Overview, Dynamics Research Corporation, no date

[AKIYAMA] F. Akiyama, "An Example of Software System Debugging," Proceedings of the IFIP Congress, 353-58(1971)

[BASILI75] V. Basili and A. Turner, "Iterative Enhancement: A Practical Technique for Software Development," IEEE Transactions on Software Engineering, Dec. 1975

[BASILI79] V. Basili and R. Reiter, Jr, "Evaluating Automatable Measures of Software Development," Proceedings of Workshop on Quantitative Software Models, Oct. 1979

[BASILI80-2] V. Basili, "Product Metrics," IEEE Tutorial on Models and Metrics for Software Management and Engineering, 1980

[BASILI80-1] V. Basili, "Changes and Errors as Measures of Software Development," IEEE Tutorial on Models and Metrics for Software Management and Engineering, 1980

[[BASILI84] V. Basili, "Software Errors and complexity: an Empirical Investigation," communications of the ACM, Jan. 1984

[BEIZER83] B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold Co., 1983

[BEIZER84] B. Beizer, *Software System Testing and Quality Assurance*, Van Nostrand Reinhold Co., 1984

[BISANT] D. Bisant and J. Lyle, "A Two-Person Inspection Method to Improve Programming Productivity," IEEE Transactions on Software Engineering, Oct. 1989

[BOEHM76] B. Boehm, J. Brown, and M. Lipow, "Qualitative Evaluation of Software Quality," Proceedings of Second International Conference on Software Engineering, IEEE, 1976

[BOEHM81] B. Boehm, *Software Engineering Economics*, Prentice-Hall, 1981

[BOEHM88] B. Boehm, "A Spiral model of Software Development and Enhancement," IEEE Computer, May 1988

[BOOCH] G. Booch, *Software Engineering with Ada*, Benjamin/Cummings Publishing Co,, 1983

[BROOKS] F. Brooks, Jr., *The Mythical Man-Month*, Addison-Wesley Publishing Co., 1975

[CALDIERA] G. Caldiera and V. Basili, "Identifying and Qualifying Reusable Software Components" IEEE Computer, Feb. 1991

[COBB] R. Cobb and H. Mills, "Engineering Software under Statistical Quality Control," IEEE Software, Nov. 1990

[CURTIS] B. Curtis, S. Sheppard, and P. Milliman, "Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics," Proceedings, 4th International Conference on Software Engineering, IEEE, 1976

[DENSON] M. Denson and R. Hooi, "Validation Test Suite Coverage Analysis," Internal SofTech working paper, 1990

[DEUTSCH] Deutsch, M. and Willis, R., *Software Quality Engineering: A Total Technical and Management Approach*, Prentice-Hall, 1988

[FAGAN] M. Fagan, "Design and code inspections to reduce error in program development," IBM System Journal, vol. 15, no. 3, 1976

[FUNAMI] Y. Funami and M. Halstead, "A Software Physics Analysis of Akiyama's Debugging Data," Proceedings of the Symposium on Computer Software Engineering, 1976

[GAITIER] R. Gautier and P. Wallis, editors, Software reuse with Ada, IEEE Computing Series 16, Peter Peregrinus Ltd. publishers, 1990

[GANNON] J. Gannon, E. Katz, and V. Basili, "Metrics for Ada Packages: An Initial Study," Communications of the ACM, 1986

[GIBSON] Gibson, V. and Senn, J., "System Structure and Software Maintenance Performance," Communications of the ACM, March 1989

[GILB] T. Gilb, *Principles of Software Engineering Management*, Addison-Wesley Publishing Company, 1988

[HUMJ87] W. Humphrey, "Characterizing the Software Process: A Maturity Framework," Technical Report CMU/SEI-87-TR-11, June 1987

[HUMS87] W. Humphrey, "A Method for Assessing the Software Engineering Capability of Contractors," Technical Report CMU/SEI-87-TR-23, Sept. 1987

[HUM89] W. Humphrey, "The State of Software Engineering Practice: A Preliminary Report," Technical Report CMU/SEI-89-TR-1, Feb. 1989

[McCABE] T. McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, Dec. 1976

[McCALL] J. McCall, P. Richards, and G. Walters, "Factors in Software Quality," 3 Vols., NTIS AS-A049-014, 015, 055, November 1977

[MILLS71] H. Mills, "Chief Programmer Teams, Principles, and Procedures," IBM Federal Systems Division Report FSC 71-5108, 1971

[MILLS76] H. Mills, "Software Development," IEEE Transactions on Software Engineering, Dec. 1976

[MEYER] B. Meyer, "Reusability: The Case for Object-Oriented Design," Software Reusability. Vol. II, ACM Press, 1989

[MUSA75] J. Musa, "A Theory of Software Reliability and its Application," IEEE Transactions on Software Engineering, Sept. 1975

[MUSA80] J. Musa, "Software Reliability Measurement," The Journal of Systems and Software, Elsevier North Holland, Inc. 1980

[PARNAS] D. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM, Dec. 1972

[PRESS] R. Pressman, *Software Engineering: A Practioner's Approach*, McGraw-Hill Book Company, 1987

[PRIETO-DIAZ] R. Prieto-Diaz and P. Freeman, "Classifying Software for Reusability," IEEE Software, Jan. 1987

[RAPIDSTD] *RAPID Center Standards for Reusable Software*, October 1990.

[RUBEY] R. Rubey, J. Dana, and P. Biché, "Qualitative Aspects of software Validation," IEEE Transactions on Software Engineering, June 1975

[RUSSELL] G. Russell, "Experience with Inspection in Ultralarge-Scale Development," IEEE Software, Jan. 1991

[TAKAHASHI] M. Takahashi and Y. Kamayachi, "An Empirical Study of a Model for Program Error Prediction," IEEE Transactions on Software Engineering, Jan. 1989

[WITHROW] C. Withrow, "Error Density and Size in Ada Software," IEEE Software 1990

[YADAV] S. Yadav, R. Bravoco, A. Chatfield, and T. Rajkumar, "Comparison of Analysis Techniques for Information Requirement Determination," Communications of the ACM, Sept. 1988